1990

Concurrent programming :

Gary J. Stolz
Lehigh University

CONCURRENT PROGRAMMING

MODULA-2 vs. PASCAL

by

Gary J. Stolz

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1989

This thesis is accepted and approved in partial fulfillment of the requirements

for the degree of Master of Science in Computer Science

Dec. 14, 1989
(date)

Samuel L. Gulden
Professor in Charge

Lawrence J. Vanucci Jr.
Chairman of Department

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (CONT.)

ABSTRACT

This thesis will examine various aspects of concurrent programming. The features of two concurrent programming languages, Concurrent Pascal and Modula-2, will be discussed and compared. The fundamental concepts of concurrent programming, as implemented in Concurrent Pascal and Modula-2 will also be presented. Finally, Modula-2 and Concurrent Pascal solutions to the classic *Readers and Writers* problem will be presented.

INTRODUCTION

Modula and Pascal are two highly structured and strongly typed programming languages developed by Nicklaus Wirth at the Institut für Informatik of ETH - Zürich, Switzerland. The original version of Pascal was developed in the late 60's primarily from a predecessor called Algol-60. Pascal was designed as a general purpose language intended for use by Wirth's students at the ETH in Switzerland. The first implementation of Pascal was in 1970. Since then, Pascal has developed widespread popularity because of its structured approach to programming and the relatively fast speed and efficiency of its compiler. Pascal is especially popular as a teaching tool at many universities throughout the world. The design of Concurrent Pascal is generally attributed to P. Brinch Hansen of the University of Southern California. As with Pascal, Concurrent Pascal was developed at a university as an abstract programming language. Concurrent Pascal incorporates the *process*, *monitor* and *class* concepts for structured concurrent programming. The monitor is a shared data structure used by Concurrent Pascal to handle the problems generally associated with concurrent programming. Concurrently executing processes are assured mutually exclusive access to the monitor section of a program by the Concurrent Pascal runtime system.

The original version of Modula had emerged from experiments in

multiprogramming and had been implemented experimentally in 1975. Modula-2 was developed from both Modula and Pascal. Important features of both languages were included in the development of Modula-2. As a result, Modula-2 includes facilities for multiprogramming and the *module* concept (for which it was named), as well as a highly structured nature and relatively fast speed of compilation. The first implementation of Modula-2 became operational in 1979. After further use and testing, the compiler was released to the public in 1981. As with Pascal, Modula-2 is gaining popularity, but largely because it provides a powerful system design tool and requires relatively minimal operating system support. Modula-2 has no built-in mechanisms for mutual exclusion or process synchronization. Instead, it provides the user with the tools for constructing such mechanisms. The primitive procedures provided for this purpose are the *Newprocess* and *Transfer* procedures. The procedures *StartProcess, Send, Wait, Awaited* and *Init* are higher level concurrent programming facilities generally provided through the inclusion of external modules in most versions of Modula-2. Throughout the rest of this paper, *Modula* will be used as a synonym for *Modula-2.*

COMPARISON OF FEATURES IN LANGUAGE STRUCTURE

Both Modula and Pascal have sequential and concurrent programming structures. Let's first examine the sequential programming facilities of both languages.

There are many similarities between Modula and Pascal as one might expect, because many of the features of Modula were taken from Pascal. Since so many of the features are the same, we will concentrate primarily on the differences in the sequential programming structures of the two languages. Both Modula and Pascal have the *INTEGER*, *REAL*, *BOOLEAN* and *CHAR* elementary or simple data types. In addition to these, Modula also has the following elementary data types:

CARDINAL - The type *CARDINAL* is a subset of the type *INTEGER* and consists of all non-negative whole numbers.

BITSET - Values that belong to the type *BITSET* are sets of integers between 0 and N-1, where N is defined by the computer system (typically N equals the computer's word length or a small multiple of it). Constants or variables of this type are denoted as sets. Examples are {3..9}, {5}, {} and {2,4,6,8}.

The more abstract data types available in both languages (*RECORD*, *SET*,

POINTER, *SUBRANGE* and *ENUMERATION*) are also nearly identical. The only difference is that Modula allows variables of type *PROCEDURE*; however, some of the newest versions of Pascal also allow variables of this type.

Modula has a *LOOP* statement which allows a sequence of statements to be executed for an indefinite period of time. The *EXIT* statement can be used to exit the loop when a given condition occurs. Pascal does not have a corresponding structure. Concurrent Pascal has the *CYCLE* statement which is similar to the loop statement, but is intended to execute a sequence of statements forever and thus there are no means provided for exiting the cycle statement.

The concept of a **function** is constructed differently in Pascal and Modula. Pascal has an explicit *FUNCTION* statement to define a function and its operation. For example, the function definition in Pascal for a function *squareroot* is as follows:

```
FUNCTION squareroot(value : REAL) : REAL;

        (* constant and variable definitions *)

BEGIN

        (* statements executed during the function call *)

        squareroot := result

END; (* squareroot *)
```

Figure 1

Modula uses the *FUNCTION PROCEDURE* to accomplish the same purpose as the Pascal function. The function procedure is actually a variant of a simple procedure definition. The function *squareroot*, as implemented in Modula, is as follows:

```
PROCEDURE squareroot(value : REAL) : REAL;

    (* constant and variable definitions *)

BEGIN

    (* statements executed during the
    function procedure call *)

END;

RETURN result

END squareroot;
```

Figure 2

As one can see, the Pascal function and the Modula function procedure are really quite similar. The two major differences are first, the ability to use the procedure in Modula to act like a function and secondly, the way that the function result is returned to the calling program. Pascal uses the function name to return the result of the function execution and Modula uses the *RETURN* statement to return the computed result to the calling program. The Modula return statement can occur anywhere within the function procedure definition and can return any result. The Pascal function can also return any result, but it must be done by assigning that result to the function name in the last executable statement of the function definition. This gives

Modula a bit more flexibility for defining functions because of the ability to return any variable as a result to the calling program at any point within the function procedure definition.

The major difference between Modula and Pascal is the concept of the *module*. An entire Modula program is called a module, given a name and has the following format:

```
MODULE name;

<import lists>

<declarations>

BEGIN

<statements>

END name.
```

Figure 3

A module does not necessarily have to be an entire program. A module can be, for example, a set of utilities to perform needed operations, such as input and output. A module allows precise control over the availability and visibility of names and provides a mechanism for introducing new types. Modules can be kept in a program *library* and automatically referenced when a program is loaded and executed. In this way, it is possible to prepare collections of frequently used operations to avoid reprogramming them each time a program needs such operations. With modules it is possible to separately compile the source code of each module and store the resulting code

as a compiled program module in the program library. The main program can then import objects from the precompiled modules after it is linked to them at load time. The subsidiary modules can also import objects from other modules, creating an entire hierarchy of modules.

In general, a module has two primary parts, the definition part and the implementation part. The definition part contains the definitions of any exported identifiers. The identifiers may be variables, types, procedure names, etc. Variables declared in a definition module are considered global to any module that imports them. Procedure declarations in definition modules consist of a heading only. The procedure body is defined in the implementation module. An importer of any module only needs to have the definition part available. The implementation part can remain the property of the module's designer. The following is an example of a Modula definition module *Terminal*, which includes routines for handling input and output to a standard alphanumeric terminal.

```
DEFINITION MODULE Terminal;  (* by S. E. Knudsen *)

    PROCEDURE Read (VAR ch: CHAR);

    PROCEDURE BusyRead (VAR ch: CHAR); (* returns
                null if no character was typed *)

PROCEDURE ReadAgain; (* causes the last
                character read to be returned again
                until the next call of Read *)

PROCEDURE Write(ch: CHAR);
```

```
        PROCEDURE WriteLn; (* terminate line *)

        PROCEDURE WriteString (s: ARRAY OF CHAR);

END Terminal.
```

Figure 4

Both the definition and implementation part of a module may contain import lists. If a module name is imported, all identifiers are automatically imported. However, they must be qualified to avoid a conflict of names in the event two modules use the same identifier. The means of qualifying imported identifiers is like a record's field identifiers. For example, if a module X exports identifiers a, b and c through the export statement

```
EXPORT a, b, c; ,
```

then a module importing module X with the statement

```
IMPORT X;
```

can reference the imported identifiers with the designators `X.a`, `X.b` and `X.c`.

The module is particularly convenient for the establishment of program libraries because of its ability to publicize the definition part of the module and hide the implementation part. Most versions of Modula include a collection of standard routines available for all programming environments.

Until recently, Pascal had no structure analogous to the module found in Modula. However, some newer versions of Pascal have the ability to link the main program to precompiled subprograms. This is done with the *uses*

directive. For example, to include functions or procedures to handle video and sound terminals, the statement,

```
uses Crt;
```

is placed in the Pascal program before the program header. These subprograms are called *units* and consist of two parts, the interface section and the implementation section. The structure is much like the structure of modules. The interface section contains *public* or *visible* identifiers. These may consist of constant, variable or type definitions as well as procedure or function headers. The interface section can also contain *uses* statements to include other units, thus creating the same levels of hierarchy that we saw in the Modula definition module. The implementation section may also contain constant, variable and type definitions. These identifiers, however, are considered private and cannot be accessed by the calling routine. Procedures and functions with headers in the interface section of a unit are defined in the implementation section to keep their implementation private while allowing public access to the function that they perform. Procedures and functions may also be defined in the implementation section without having a corresponding header in the interface section. This keeps both their definition and usage private and only accessible by the unit in which they are defined.

An important point to note is that if two units use the same identifier and are simultaneously used by a program, the identifier defined last takes precedence over the first defined. For example, if both the *Crt* and *Terminal* units define

a variable called *tbuffer*, the *tbuffer* defined in the *Crt* unit will not be available if the following statement is used to include these two units:

```
uses Crt, Terminal;
```

Conversely, the *tbuffer* variable defined in the *Terminal* unit will not be available if the two units are included with the following statement:

```
uses Terminal, Crt;
```

As one can see, with the addition of *units* to Pascal, Modula and Pascal are really quite similar and offer many of the same programming advantages. The most significant differences between the two languages occur in the facilities each provides for concurrent programming. This leads us to the next step - a comparison of the concurrent programming features of Modula and Concurrent Pascal.

There are several types of multiprogramming systems. Among them are the following:

1. The computer consists of several identical processors. Processes are executed in genuine concurrency.

2. The computer consists of a single processor. The processor can only act on a single process at any instant. Processes are time multiplexed and scheduled for execution by a scheduler program. These processes are said to be *pseudo-concurrent.*

3. The computer consists of a single processor and can only act on a single

process at a time. The processes are not scheduled, but must explicitly be given control of the processor. These processes are sometimes called *coroutines* and are said to be *quasi-concurrent.*

We will restrict ourselves to single processor systems in this paper; however, the principles can, in general, be applied to multi-processor systems as well.

Modula has no built-in mechanisms for mutual exclusion or process synchronization - two necessary conditions for concurrent programming. Instead, Modula provides the user with the means for constructing such mechanisms as he sees fit for a given situation. Modula has two low level procedures for providing the basic underlying mechanisms for concurrent programming. These procedures are located in the standard *SYSTEM* module. In particular, these are the procedures *NEWPROCESS* and *TRANSFER*. The SYSTEM module also exports the types ADDRESS and PROC as mechanisms for concurrent programming. The type ADDRESS is used for variables that store destination or return addresses. An ADDRESS is, in fact, defined as type *POINTER TO WORD*, where a WORD is defined by the word size of the computer. The type PROC corresponds to a parameterless procedure that will be executed as a coroutine.

The procedure NEWPROCESS is called in order to create a coroutine, and is declared as

```
PROCEDURE NEWPROCESS(P:PROC; A:ADDRESS; n:CARDINAL;
                     VAR new:ADDRESS);
```

where **P** is the program to be executed as a coroutine. **A** is the address of the workspace allocated to store the local variables of the coroutine and the coroutine's state while it is suspended. **n** is the size of the workspace in storage units and **new** is a pointer to the place where the execution of the coroutine P shall begin when control is transferred to it. The heading of the Transfer procedure is

```
PROCEDURE TRANSFER(VAR source, destination: ADDRESS);.
```

A call of the Transfer procedure causes the *source* to be suspended and the *destination* to be resumed at its current point of suspension. When control is returned to the source, execution will resume at the point immediately following the TRANSFER statement. Thus, coroutines can be created, started, terminated and executed explicitly by using the NEWPROCESS and TRANSFER procedures.

In order to provide a higher level of abstraction for multiprogramming, Modula provides the standard module *Processes.* The *Processes* module includes the type *SIGNAL* along with these procedures - *StartProcess, SEND, WAIT, Awaited* and *Init.*

The procedure *StartProcess* is declared as

```
PROCEDURE StartProcess(P:PROC; n:CARDINAL);
```

where *P* is the name of a procedure which is executed as a process and *n* is the number of words to assign for P's workspace. A variable declared as type SIGNAL serves to synchronize processes. A process can only perform two

operations on a SIGNAL: a process may send a signal and it may wait for a signal. A signal is sent as an indication that a certain condition has arisen. When a signal is sent, a process waiting for that signal may resume execution. A signal can awaken only a single process.

The SEND procedure is defined as

```
PROCEDURE SEND(VAR s:SIGNAL);
```

and is used simply for sending a specified signal. If no process is waiting for the signal, this is considered a null operation.

The WAIT procedure is declared as

```
PROCEDURE WAIT(VAR s:SIGNAL);
```

and is used to suspend the operation of a process until the signal *s* is sent. Processes waiting for a signal s are placed in a FIFO queue and reactivated in that order when the signal s is sent.

The Awaited procedure is actually a procedure function and is declared as

```
PROCEDURE Awaited(VAR s:SIGNAL): BOOLEAN;
```

The Awaited procedure will return a value of TRUE if any process is waiting for the signal s to be sent. It will return FALSE otherwise. This procedure is primarily used to test if any processes are waiting for a specific signal, so that unnecessary signals will not be sent.

The Init procedure is declared as

```
PROCEDURE Init(VAR s:SIGNAL);
```

and is used to perform the compulsory initialization of a signal. The last concurrent programming feature of Modula is the *monitor.* A monitor is a data structure which guarantees mutual exclusion of processes and can thereby ensure the integrity of its local data. A monitor provides mutual exclusion by ensuring that execution of a calling process will be temporarily delayed while another process is executing any one of the monitor's procedures. By specifying a priority in the heading of a module declaration, a module is designated to be a monitor. For example

```
MODULE Buffer[2];
```

designates the Buffer module as a monitor. Any cardinal number can be used to indicate that a module is to be a monitor.

Now, let's examine the concurrent programming mechanisms used by Concurrent Pascal. There are three primary mechanisms for implementing concurrent programming. They are the *process*, *monitor* and *class.* All three are special data types called *system types.*

The type *process* defines a sequential program. A process consists of a private data structure and the sequential program that operates on it. A process cannot operate on the private data of any other process. An example of a process is an endless cycle that fills a buffer with data and outputs it to a printer. The process definition for such a buffer would be similar to the following:

```
TYPE printerprocess = PROCESS(buffer: linebuffer);
```

```
(* local variable definitions *)

BEGIN

(* sequential program statements *)

END;
```

Figure 5

In the main program, the following variable definitions would serve to declare a process *printer* that acts on a buffer called *inbuffer*.

```
VAR inbuffer: linebuffer;

printer: printerprocess;
```

The printer process is started by an *init* statement

```
init printer(inbuffer);
```

which also serves to allocate storage for the private variables of the process. The init statement is used to start concurrent execution of any number of processes and also to define their access privileges. In the init statement above, the printer process is given access to the input buffer by using the variable *inbuffer* as a parameter. A process is only able to access its own parameters and private variables. Variables that are accessible to a system component (a variable of type process, monitor or class) are those that are declared within its type definition. This access rule and the init statement make it possible for a programmer to explicitly state accessibility rights and have them checked by the compiler.

The *monitor* defines a set of *shared variables.* Within a monitor, variables of type *queue* are used to delay processes attempting to gain entry into the monitor when another process already has control of the monitor. Monitor procedures are marked with the word *entry* to distinguish them from local procedures used within the monitor. The monitor's local procedures cannot be accessed by processes entering the monitor. The primary use of entry procedures is to delay calling processes. A process is delayed by using the *delay* statement. The statement

```
delay(sender);
```

will delay the calling process and place it in a waiting queue called sender. The *continue* statement is used in conjunction with the delay statement to restart suspended processes. The statement

```
continue(sender);
```

will restart the process waiting in the sender queue (if any). Every monitor must be initialized with an *init* statement. The init statement allocates storage for the shared variables of the monitor and executes the *initial statement.* The initial statement is the executable part of a monitor not contained within any procedure or function definitions within the monitor itself. Once initialized, the shared variables of a monitor exist forever and are called *permanent variables.* During execution of a monitor, the monitor has exclusive access to the permanent variables in the monitor. Any processes attempting to simultaneously access a procedure within a monitor will

necessarily be executed one at a time. To prevent the occurrence of deadlocks, the following rules have been imposed on monitor calls in Concurrent Pascal. A routine must be declared before it is called. Routine definitions cannot be nested and cannot call themselves. A system type cannot call its own routine entries.

Concurrent Pascal uses delay operations to suspend the execution of a calling process for a period of time. Delay operations are performed using variables of type *queue*. Although these variables are defined as type *queue*, they are not real queues. The Concurrent Pascal queue is a single process queue. The queue is either empty or non-empty and initially it is empty. Therefore, only one process at a time can wait in the queue. Any process put in the wait queue loses access to the monitor's shared variables until another process calls the same monitor and executes a continue statement on the queue in which the suspended process is waiting. The continue operation causes the process executing the operation to exit the monitor procedure and allows a process waiting in the selected queue (if any) to resume execution of the monitor procedure at the point where it was suspended. One final operation on a queue is the boolean function *empty*. The statement

```
empty(sender);
```

will return a value of TRUE if no process is waiting in the sender queue; it will return FALSE otherwise.

The final system type we will present here is the type *class*. A class is a system component that cannot be called simultaneously by processes or monitors. To guarantee this condition, the following rule must be imposed on the class type. A class must be declared as a permanent variable within a system type and can only be passed as a permanent parameter to another class (but not to a process or monitor). This implies that a nested chain of class calls can only be started by a single process. Consequently, simultaneous class calls cannot occur at run time and require no need for scheduling their execution.

INPUT/OUTPUT MECHANISMS

Peripheral devices can often be a potential source of erratic behavior in any system. Many of the input/output operations associated with peripherals are interrupt driven and deserve careful attention in multi-programming environments.

In this section, we will examine three primary schemes for handling input and output in a multi-programming environment. They are

1. Busy Waiting I/O

2. Interrupt Driven I/O

3. Direct Memory Access I/O

An example of busy waiting I/O is the intercommunication between a computer and the keyboard and screen of a terminal device. When a character is to be written to the screen, the system must first wait until the interface is ready to receive it. If the transfer of a previous character is still in progress, obviously the interface is not in a ready state. The screen can advertize its status by using a status bit that can be set to "1" when the screen is ready to accept input. Similarly, the keyboard can use a status bit to indicate its current state. By using this method, it is possible to restrict, to one, the number of processes that can send output to a terminal. If the

terminal is in a busy state (status bit set to "1"), any process attempting to send output to the terminal will be suspended and put in a waiting state. A major disadvantage of busy-waiting I/O is that the processor spends most of its time waiting for the ready bits of the status register to be set. This may be tolerable if there is only one terminal to service and there is not much processing to be done. However, if the system is a multi-user system or has heavier amounts of processing to perform, this method of performing I/O would be highly inefficient.

The second method for performing I/O is *interrupt driven I/O.* Interrupt driven I/O is a method in which the processor can be relieved of the task of testing for the ready status of every peripheral device. The peripherals are said to be in *Interrupt mode.* In this case, an interrupt can be regarded as a procedure call initiated by some external event unrelated to the central processor. Interrupts are usually relayed to the processor by means of an interrupt vector. Each source of an interrupt has its own interrupt vector located at a fixed address in the computer's main memory. An interrupt vector typically contains the following information:

- The address of a procedure to be called in response to the interrupt, i.e. a new address to be loaded into the program counter.
- The new value to put into the processor's status register, which will define the processor's priority during the handling of the interrupt.

When an interrupt occurs, the contents of the program counter and the processor status register are saved on a stack. These registers are then loaded with the values passed in through the interrupt vector. Interrupts from different sources will, of course, have differing values in their interrupt vectors. The effects of handling the interrupt vector are to change the processor priority and to start execution of the interrupt handling procedure. After the interrupt has been served, the old values of the program counter and the processor status register are restored from the stack. This causes the processor to return to its previous priority and the execution of instructions is resumed at the point where the interrupt occurred. Interrupt driven I/O works well for relatively slow peripheral devices, such as printers, but would be intolerable for high speed devices like a hard disc.

This leads us to the third and final mechanism for performing I/O - *Direct Memory Access (DMA) I/O.* A device, such as a hard disc, that requires that large amounts of data be transferred in relatively little time, cannot operate efficiently in a mode where only a single byte of data is transferred at a time. The operation of transferring the byte would scarcely have been completed before the next byte of data presented itself. The DMA I/O mechanism is one which can transfer complete blocks of words. The DMA I/O interface typically requires four registers to specify the following:

- A device access address.

- A main memory address.
- The number of words to be transferred.
- The direction of data transfer (e.g. disc to memory or memory to disc).

An interrupt is generated when the DMA interface has completed the data transfer.

PROCESSES

We have already examined the implementation of processes in Section 2 of this paper. Now, let's take a more detailed look at processes by considering an operating system whose aim is to provide a fair service to each of the users connected to the system. The system must be capable of receiving commands entered by each user, executing them and returning the results to that user. If we assume that the common I/O device for a user is a terminal, then for any terminal in the system, we can express the I/O operation as follows:

```
PROCEDURE Serve_Terminal(no : INTEGER);

BEGIN

    read a command from terminal no;

    execute this command;

    return results to terminal no;

END Serve_Terminal;
```

Figure 6

One naive way to express the time sharing for multiple terminals (in this case 3) would be as follows:

```
PROGRAM Time_sharing_system;

BEGIN

    CYCLE
```

```
                Serve_Terminal(1);

                Serve_Terminal(2);

                Serve_Terminal(3);

            END

        END;
```

Figure 7

The implementation shown above would not be satisfactory for several reasons.

- The I/O is handled through the busy-waiting mechanism. This proves to be highly inefficient since the time spent waiting for a character from terminal 1 could be used to execute commands received from terminal 2 or 3.

- The algorithm does not consider the different execution times of commands received from the various terminals. This results in inequitable treatment of the terminals.

- The possibility of different users working at different speeds is not considered in the algorithm.

To avoid the problems listed above, the I/O could be handled through interrupts. I/O handling through interrupts cannot be applied to the algorithm shown above because it is sequential in nature. An interrupt handling program cannot be sequential because the input and output must be

dealt with intermittently and for varying durations. The difficulty in handling interrupts can be overcome with the process concept.

A process by itself can be regarded as a sequential program. Specifying the simultaneous execution of more than one process defines a task that is essentially non-sequential or concurrent. From the example above, we could define a process for a terminal, n, in Concurrent Pascal as follows:

```
TYPE term_process = PROCESS(buffer : linebuffer);

BEGIN

    CYCLE

        serve_terminal(n);

    END

END;
```

Figure 8

In a genuinely concurrent environment, each process would execute independently on its own processor. On a single processor system, each process is executed for only a short period of time on a regular basis, giving the impression that each process has its own processor. As we have discussed previously, the two types of non-genuine concurrency are:

- *Pseudo-concurrency* - Only one process executing at any instant and process switching is controlled outside of the control of the processes themselves.

- *Quasi-concurrency* - Only one process is executing at any instant and process switching occurs at the request of the active process (the one executing).

Process switching in pseudo-concurrency is done by a set of procedures called the *kernel.* The kernel is simply a program that allocates time among active processes. The kernel must be capable of handling a data structure called the *process descriptor.* The process descriptor for a particular process provides all relevant information for that process. This includes:

- The variables belonging to the process.
- The priority and status of the process.
- The contents of the process's registers during the times when it is suspended.

In order to determine when to switch processes, the kernel uses a clock which produces interrupts at regular intervals (typically 50 times per second). The process that is active when an interrupt occurs, is suspended in favor of another one. In general, the sequence of events that occur during the process switching is as follows:

1. The process registers for the interrupted process are saved on top of its stack.
2. The stack pointer is saved in the process descriptor.

3. The process descriptor is placed at the end of a linked list that contains process descriptors of other suspended processes. Execution of a process resumes when its descriptor has worked its way back to the head of the linked list.

4. When a process is to resume execution, the stack pointer is loaded with the value of the descriptor at the beginning of the linked list. The stack of the new process can now be recovered and the kernel can allocate the processor to the new process.

5. Finally, the values from the new process's stack are loaded into the process registers. Execution of the new process now begins.

It is also possible for interrupts to be generated by sources other than the process clock. These interrupts may also lead to a process switch.

MUTUAL EXCLUSION

On many computer systems there is a need for shared resources, such as printers, discs, etc. As many processes attempt to access these resources, there must be a system in place to assure that the resources that are sequential in nature are only accessed by one process at a time. Consider the case of two processes simultaneously attempting to write information to a terminal. Take, for example, a printer error and a disc error occurring and the two processes reporting these errors each attempting to write their error messages to the user's terminal at the same time. Letting both processes write to the terminal simultaneously would certainly cause the messages to be garbled and unintelligible. There must be some means of keeping all other processes from writing to a terminal if another process has already begun writing to the terminal. This procedure of allowing only one process at a time to access a shared resource or shared data is known as *mutual exclusion.* The types of resources that require mutual exclusion are called *critical resources* and the part of any program that accesses these resources is known as the *critical section.* To work correctly, a critical section must guarantee that only one process can be executing between the beginning and end of the critical section. In addition, only one process at a time must be allowed to enter the critical section, only if no other process is currently occupying it. A process

seeking to enter the critical section must be able to do so within a finite period of time (i.e., a process waiting to enter the critical section cannot be delayed indefinitely by another process attempting to enter the critical section). There are several methods available to handle mutual exclusion in critical section. They are:

- Busy-waiting
- Masking Interrupts
- Locks
- Semaphores

We will now examine each of these methods.

The idea behind *Mutual Exclusion through Busy-waiting* is to have processes waiting to access a resource sit idle in a busy loop until the resource becomes available. This can be done by using a boolean variable to indicate if any process currently has control of a resource. If any process does *own* a resource, no other process will be allowed to access the resource until the owner process relinquishes control. In time sharing systems, a problem can occur if a process attempting to access a resource is suspended before it is able to set the boolean variable to a state that prevents other processes from entering the critical section. If this occurs, there would be more than one process in the critical section at the same time. To avoid this problem, a boolean variable can be assigned to each process for each resource. A process

attempting to gain access to a critical section will wait in a busy loop until there are no other requests by other processes for entry into the critical section. In order to prevent a process from waiting indefinitely for entry into a critical section, there must be a mechanism to assure that each process will get a chance for entry into the critical section. As an example of a critical section procedure for two processes, consider the following Pascal monitor procedure called *start*:

```
PROCEDURE ENTRY start(no : noprocess);

BEGIN

    request[no] := true;
    (* makes request to enter the critical section *)

    turn := 3 - no; (* number of the other process *)

REPEAT

UNTIL (NOT request[3 - no]) OR (turn = no);

END;
```

Figure 9

This example can, of course, be generalized for n processes. The variable *turn* is used to decide which process will be allowed to enter the critical section if more than one process is attempting to enter the critical section simultaneously. It also guarantees that no process will be suspended indefinitely due to repeated requests by other processes. This is a valid method to assure mutual exclusion but is not realistic for multiuser systems. The busy-waiting method results in far too much waste of processor time and

is unacceptable in real time situations.

As we have discussed in the last section, process switching can occur because of interrupts. Mutual exclusion can be achieved by masking interrupts (i.e., processing of certain interrupts does not occur) so that no more process switching occurs. This guarantees that the mutual exclusion requirements for entry into a critical section will not be violated. Consider the following example of a Modula process to access the console of a computer system.

```
PROCEDURE p1;

BEGIN

    disable_interrupts;

    console.write("DISC ERROR ...");

    enable_interrupts;

END p1;
```

Figure 10

This example for implementing mutual exclusion has two disadvantages.

- Additional I/O (such as input from a terminal) is prohibited during interrupt mode in the critical section.

- There is a risk that interrupts will be lost if the critical section is long (e.g., a second interrupt from the same device occurs before the first has been processed).

Now let's consider the case of a multiprocessor system where several of the

processors share a common memory. Mutual exclusion is not guaranteed by simply masking interrupts, because of the fact that different processes are actually executing in parallel. This problem can be solved by designating a location in the shared memory to indicate whether any process is in a critical section. Before a process enters a critical section, it must check this location in memory to see if any other process is currently in a critical section. If the process finds that another process is in a critical section, it must wait until the memory location has changed to indicate that no other process is in a critical section. The waiting process can then set the memory location appropriately and proceed with its critical section. Upon exit from the critical section, the process must reset the memory location to the free state. A Pascal function similar to the following can be used to test the memory location and set its state to *busy*:

```
FUNCTION test_and_set(VAR v : state) : state;

BEGIN

    test_and_set := v;

    v := busy;

END;
```

Figure 11

A process attempting to enter a critical section would execute the following statements before gaining access to the critical section

```
REPEAT
```

```
UNTIL test_and_set(v) = free;
```

and then execute the following after exiting the critical section

```
v := free;.
```

The next method we will discuss for assuring mutual exclusion is *Mutual Exclusion Using Locks.* A lock is defined as a variable of the following type:

```
TYPE lock_type = RECORD
                    state : (open, closed);
                    waiting = list of processes;
END;
```

Figure 12

A lock can be in one of two states - open or closed. It may also consist of a list of suspended processes waiting for the lock to be changed to the open state. A process can act on a lock through one of two procedures defined as follows:

```
PROCEDURE lock(VAR v : lock_type);
BEGIN
   IF v.state = open
   THEN v.state := closed
   ELSE (* suspend process in list 'v.waiting' *);
END;
PROCEDURE unlock(VAR v : lock_type);
BEGIN
```

```
        IF list 'v.waiting' NOT empty
        THEN awaken a process
             of list 'v.waiting'
        ELSE v.state := open;
    END;
```

Figure 13

The calls of these procedures are placed around the execution of a critical section in a program as in the following Modula process:

```
PROCEDURE p1;
BEGIN
    lock(v);
    console.write("DISC ERROR");
    unlock(v);
END p1;
```

Figure 14

The lock and unlock procedures are critical sections also and should be placed within a Concurrent Pascal or Modula monitor. The procedures lock and unlock are known as mutual exclusion *primitives.* This means simply, that during execution of those sections of code, the current process will not lose control of the processor before completing the execution.

The final mechanism that we will discuss for providing mutual exclusion is the *semaphore.* As we have just seen, locks are appropriate for handling simple

mutual exclusion problems, but may not be adequate for more complicated applications. A problem, such as allocating *m* printers to *n* processes, where n > m, can be handled by the semaphore. A semaphore is a variable of the following type:

```
TYPE semaphore = RECORD

                    n : INTEGER;

                    waiting : list of processes;

END;
```

Figure 15

The rules for using semaphores are very simple. No process can directly access the fields of a semaphore. Two primitives, *P* and *V*, are used for handling access to a semaphore. These primitives are defined as follows:

```
PROCEDURE ENTRY P(VAR s : semaphore);

BEGIN

    s.n := s.n - 1;

    IF s.n < 0 THEN

        block the calling process and

        place it at the tail of list 's.waiting'

    END

END;

PROCEDURE ENTRY V(VAR s : semaphore);

BEGIN
```

```
        s.n := s.n + 1;
        IF s.n <= 0 THEN
            awaken the process at the head
            of list 's.waiting'
        END
    END;
```

Figure 16

The procedures *P* and *V* are defined here as Concurrent Pascal monitor entry procedures because they are considered critical sections and must be handled accordingly. The order in which semaphores are invoked is important. Care must be taken to assure that semaphores are not invoked in the wrong order. This can result in a deadlock situation in which two processes are blocking each other, preventing either from continuing execution.

COMMUNICATION BETWEEN PROCESSES

In this section, we will briefly discuss three mechanisms for providing communication and synchronization between processes. They are:

- Synchronization using events
- Synchronization using semaphores
- Synchronization using monitors

The *event* is the simplest tool for providing cooperation among processes. Events are variables of the following type:

```
TYPE event = RECORD
                occurred : BOOLEAN;
                waiting : list of processes;
             END;
```

Figure 17

The boolean field of an event, *occurred*, describes the two possible states of an event. An event has either occurred (occurred = TRUE) or it has not (occurred = FALSE). The second field of an event, *waiting*, is a list of processes that are waiting for the event to occur. An event can be manipulated by one of three operations - *wait*, *trigger* and *reset*. These operations are defined by the following procedures:

```
PROCEDURE wait(VAR e : event);

BEGIN

    IF NOT e.occurred

    THEN block process in the list 'e.waiting';

END;

PROCEDURE trigger(VAR e : event);

BEGIN

    e.occurred := TRUE;

    awaken all processes in the list 'e.waiting';

END;

PROCEDURE reset(VAR e : event);

BEGIN

    e.occurred := FALSE;

END;
```

Figure 18

Synchronization using semaphores is a mechanism for providing cooperation among processes by considering the semaphore to be a generalization of the event. The semaphore consists of two procedures: *P*, which decrements an integer variable and blocks the calling process if the value becomes negative and *V*, which increments the same integer variable and awakens a waiting process, if any. These procedures are defined as follows:

```
PROCEDURE P(VAR s : semaphore);
```

```
BEGIN

    s.n := s.n - 1;

    IF s.n < 0

    THEN block calling process in list 's.waiting'

END;

PROCEDURE V(VAR s : semaphore);

BEGIN

    s.n := s.n + 1;

    IF s.n <= 0

    THEN awaken process at head of list 's.waiting'

END;
```

Figure 19

The final mechanism for synchronizing processes that we will discuss is the *monitor*. As we have stated before, a monitor is a group of procedures and variables that can only be accessed by one process at a time. For example, if process p1 is executing procedure p of a monitor m, then a process p2 attempting to execute procedure p of the same monitor will be temporarily blocked. Synchronization within a monitor can be expressed by using *signals* along with the procedures *send* and *wait*. The effect of calling the procedure s.wait, where s is a signal, is to unconditionally suspend execution of the current process. The suspended process will remain suspended until a signal of type s is sent. To send a signal s, the procedure s.send is executed. If more

than one process is waiting for the signal s, only a single process will be awakened. The awakened process will be the one at the head of the signal's wait queue. When a process has control of a monitor and it becomes blocked by the execution of s.wait, the mutual exclusion on the monitor is released. This is an important property of the monitor and solves the problems associated with using events and semaphores for interprocess communication. The problem is the chance that the state of an event or semaphore may change between lines of a program acting on that state.

THE READERS AND WRITERS PROBLEM

In this section, we will discuss the *Readers and Writers* problem and possible solutions to the problem as implemented in Concurrent Pascal and Modula. The readers and writers problem is the problem faced by any system in which more than one process attempts to read from or write to a common resource.

In general, the solution to the problem must assure the following:

- Any number of processes can read data from the resource as long as no process is currently writing to the resource or waiting to write to the resource.

- If any process is writing to the resource, other processes attempting to read from or write to the resource must wait until the current writer process has completed the write operation.

- When the resource becomes free, processes waiting to write to the resource will be awakened one at a time and allowed to write to the resource. When all waiting writer processes have completed, then all waiting reader processes will be allowed to continue.

A possible solution to the problem is the following Concurrent Pascal monitor:

```
TYPE rw = monitor;

VAR nbr_readers : INTEGER;
```

```
        someonewriting : BOOLEAN;
        readers, writers : queue;
PROCEDURE ENTRY start_read;
BEGIN
    IF someonewriting OR NOT empty(writers)
    THEN delay(readers);
    nbr_readers := nbr_readers + 1;
    continue(readers);
END;
PROCEDURE ENTRY end_read;
BEGIN
    nbr_readers := nbr_readers + 1;
    IF nbr_readers = 0
    THEN continue(writers);
END;
PROCEDURE ENTRY start_write;
BEGIN
    IF nbr_readers > 0 OR someonewriting
    THEN delay(writers);
    someonewriting := TRUE;
END;
PROCEDURE ENTRY end_write;
```

```
BEGIN
    someonewriting := FALSE;
    IF NOT empty(readers)
    THEN continue(readers)
    ELSE continue(writers);
END;
BEGIN
    someonewriting := FALSE;
END;
```

Figure 20

In order to use the monitor procedures above, calls to the procedures *start_write* and *end_write* should be placed around the statements of a program where an attempt is made to write to a shared resource. Similarly, calls to *start_read* and *end_read* should be placed around program statements that attempt to read from a shared resource. In Modula, the use of calls to such procedures is the same. A possible Modula solution to the readers and writers problem is as shown below:

```
MODULE rw[4];
IMPORT SYSTEM, Process;
VAR readers : INTEGER;
    someonewriting : BOOLEAN;
    readingallowed, writingallowed : Process.SIGNAL;
```

```
PROCEDURE Beginreading;

BEGIN

    IF someonewriting OR Process.Awaited(writingallowed)

    THEN Process.WAIT(readingallowed);

    END; (* IF *)

    readers := readers + 1;

    Process.SEND(readingallowed);

END Beginreading;

PROCEDURE Finishedreading;

BEGIN

    readers := readers - 1;

    IF readers = 0

    THEN Process.SEND(writingallowed);

    END; (* IF *)

END Finishedreading;

PROCEDURE Beginwriting;

BEGIN

    IF readers > 0 OR someonewriting

    THEN Process.WAIT(writingallowed);

    END; (* IF *)

    someonewriting := TRUE;

END Beginwriting;
```

```
PROCEDURE Finishedwriting;

BEGIN

    someonewriting := FALSE;

    IF Process.Awaited(readingallowed)

    THEN Process.SEND(readingallowed)

    ELSE Process.SEND(writingallowed);

    END; (* IF *)

END Finishedwriting;
```

Figure 21

In the Modula solution, two signals, *readingallowed* and *writingallowed*, are used to signal when a process waiting to read or write may continue. The boolean variable *someonewriting* is used to indicate if any process is currently writing to the shared resource.

As one can see, the two solutions are quite similar. The difference lies in the use of signals by Modula to reactivate waiting processes. In Modula, the programmer must explicitly send a signal to awaken a suspended process. Concurrent Pascal uses the continue statement for restarting suspended processes. It is also important to realize that the actual read and write operations to the shared resource should be considered critical sections. As such, these processes should not be suspended by any external sources, such as a process scheduler or an interrupt, before the read or write operation has been completed.

SUMMARY

Pascal and Modula offer much the same facilities for sequential programming. The major difference between the two languages is the module, but as we have discussed, some newer versions of Pascal contain the data structure called the unit. The module and the unit provide the user with the ability to define program libraries that can be separately compiled and stored for inclusion in other programs without having to recompile the library programs.

The real notable differences are in the non-sequential programming mechanisms found in Modula and Concurrent Pascal. There is no data structure called a process in Modula. A process is defined simply as a procedure containing a sequential program. Modula has no kernel and therefore is not in charge of sharing the processor among processes. Process switching must be programmed. Modula supplies the user with the necessary low level facilities for controlling the creation of and switching among processes, but in general leaves the details of how and when this is done up to the user. On the other hand, Concurrent Pascal provides the user with the mechanisms to define processes, start processes, delay processes and restart processes, but control over how and when this is done is invisible to the user.

Both Modula and Concurrent Pascal provide the user with data structures,

such as the monitor, to give the user the ability to provide for mutual exclusion of processes during execution of critical sections of code. These data structures also serve to provide interprocess communication and synchronization.

In conclusion, we have seen that because of the nature of the concurrent programming mechanisms of Modula and Concurrent Pascal, each language lends itself to different types of applications. Modula, for example, is better suited for designing low level types of programs, such as operation systems. Concurrent Pascal is more appropriate for higher level applications, such as printer spoolers and terminal handling routines. As of the date of this paper, there are several commercially available Modula compilers. There is currently no standard for Modula and therefore the module libraries included with some of the compilers are not compatible. This will cause problems with portability of Modula software. There is a standard for sequential Pascal and it is readily available; however, Concurrent Pascal is not yet widely available commercially nor is it standardized.

REFERENCES

and

BIBLIOGRAPHY

Ben-Ari, M., *Principles of Concurrent Programming*, Prentice - Hall, Englewood Cliffs, NJ, 1982.

Brinch Hansen, P., *A Keynote Address on Concurrent Programming*, Computer Science Department of the University of Southern California, June 1978.

Brinch Hansen, P., *The Architecture of Concurrent Programs*, Prentice - Hall, Englewood Cliffs, NJ, 1977.

Brinch Hansen, P., *The Programming Language Concurrent Pascal*, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975.

Cooper, J. W., *Introduction to Pascal for Scientists*, John Wiley & Sons, New York, 1981.

Grogono, P., *Programming in Pascal*, Addison Wesley, Reading, MA,

1980.

Hartmann, A. C., *A Concurrent Pascal Compiler for Minicomputers*, Springer - Verlag, New York, 1977.

Holt, R. C., *Concurrent Programming with Operating Systems Applications*, Addison - Wiley, Reading, MA, 1978.

Schiper, A., *Concurrent Programming*, Halsted Press, New York, 1989.

Smedema, C. H.; Medema, P.; Boasson, M., *The Programming Languages Pascal, Modula, Chill & Ada*, Prentice - Hall International, Englewood Cliffs, NJ, 1983

Wirth, N., *Programming in Modula - 2*, Springer - Verlag, New York, 1985.

Wood, S., *Using Turbo Pascal Version 5*, Osborn McGraw - Hill, New York, 1989.

VITA

Gary J. Stolz was born in Northampton, Pennsylvania, on December 6, 1958. He is the son of Mr. and Mrs. John A. Stolz, Jr. He graduated from Lehigh University in 1981 with a Bachelor of Science degree in Computer Engineering. Currently, he is a Planning Engineer in the Quality Assurance organization at AT&T - Microelectronics in Allentown, Pennsylvania. His current assignments include the design and development of databases and other computer software used for the analysis and reporting of Quality data collected throughout AT&T - Microelectronics.